QuBE7.0

SYSTEM DESCRIPTION

Enrico Giunchiglia enrico.giunchiglia@unige.it
Paolo Marin paolo.marin@unige.it
Massimo Narizzano massimo.narizzano@unige.it
DIST - Università di Genova
Viale Causa 13, 16145 Genova
Italy

Abstract

In this paper we outline QuBE7's main features, describing first the options of the preprocessors, and then giving some details about how the core search-based solver (*i*) performs unit and pure literal propagation; and (*ii*) performs the "Conflict Analysis procedure" for non-chronological backtracking, generalised from the SAT to the QBF case. We conclude with the experimental evaluation, showing that QuBE7.0 is the a state-of-the-art single-engine QBF solver.

KEYWORDS: *QBF, solver, preprocessor*

Submitted March 2010; revised May 2010; published August 2010

1. Introduction

Quantified Boolean Formulas are a powerful extension of the Satisfiability (SAT) problem in which variables are universally as well as existentially quantified. QuBE is a state-of-the-art QBF solver, and since its advent, it has been kept up-to-date for solving more and more complex problems. In the last QBF Evaluations it has always been in the top-ranked solvers. In this paper we outline the latest QuBE's features, starting from the techniques that can be enabled/disabled during the preprocessing phase. We then move on to the core search-based solver, giving some details about the differences from the older QuBE6.x version, which are mainly (*i*) in the algorithm for propagating unit and pure literals; and (*ii*) in the "Conflict Analysis procedure" [12, 11] that we generalised from the SAT to the QBF case, performed to compute the asserting clause (resp. term) when it non-chronologically backtracks from a conflict (solution). We conclude with the experimental analysis in which, after fixing the bugs found during the latest evaluation, we ran QuBE7 on the same test-bed of QBFEVAL10, showing that QuBE is still a very effective, mono-engine, sequential QBF solver. QuBE7 is available at `http://www.star-lab.it/qube`.

2. QuBE7

In the QBF Evaluation 2008, QuBE6.1 [6] resulted the most powerful mono–engine QBF Solver: it ran hors concourse (since the evaluation was run by people in our same Department) and was able to solve more than twice the number of instances solved by the second solver in the rank [7]. QuBE7 is the natural evolution of QuBE6.1. It is the composition

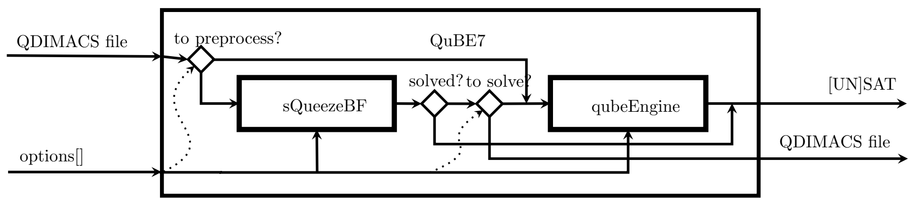

Figure 1. QuBE7 framework

of two different reasoning tools: the preprocessor sQueezeBF [9] and the new search-based core solver qubeEngine.

The framework of QuBE7 is shown in Fig. 1: given in input a Conjunctive Normal Form (CNF) QBF (`QDIMACS` format, see [7]) and some optional parameters; QuBE7 can return either the value of the input QBF or an equi-satisfiable preprocessed version of it (if the user selects to run only the preprocessor). The input parameters allow the user to enable/disable the built-in preprocessor (i.e. the user can choose to solve the instance without preprocessing), and to select which techniques of the preprocessor to use (see § 2.1). In the future, it will also be possible to select by command-line other options, such as the heuristic function qubeEngine calls during the search, whether to enable or not the non-chronological backtracking and learning from solution, and so on.

2.1 Preprocessor

A detailed description of sQueezeBF techniques can be found in [9, 10]. In Fig. 2(a) we show the main algorithm of sQueezeBF. sQueezeBF takes as input a QBF φ, and returns a simplified QBF, that may be empty (i.e., equivalent to TRUE) or contain an empty clause (i.e., equivalent to FALSE). sQueezeBF starts saving the current state of the formula at line 2, and then it applies four operations sequentially, i.e., Simplify(φ), Eq-Subs(φ), Eq-Rw(φ), Q-resolution(φ). The process is repeated until no further simplification is possible (line 9).

Simplify (line 3) gets as input the formula and simplifies it propagating all the unit and pure literals. Moreover, it also eliminates *subsumed* clauses —i.e., clauses that are a superset of another clause in φ, see [15])— or *self-subsumed* clauses, as in [10, 3].

Eq-Subs (line 4) looks first for patterns of clauses that correspond to the definitions of the output value of logical gates AND, OR, and XOR; then it sorts their symbolic definition building a forest in which each node is a defined variable and the edges connect each defined variable to the variables defining it. The defined variables (together with their definitions) can then be removed from the formula by substituting them with the right hand side of the definition. This process starts from the roots of the forest, excluding the defined variables whose substitution causes an increase of the size of the formula. This technique was introduced for SAT in [3].

Eq-Rw (line 5) processes the equivalences that are discarded by Eq-Subs because leading to an increase of the size of the formula. Briefly put, its strategy is to re-encode the original (Tseitin [14]) definition into a new encoding related to the one introduced by Plaisted in [13] (see [10] for more details).

```
0 function sQueezeBF($\varphi$)
1     do
2        $\varphi$' = $\varphi$
3        $\varphi$ = Simplify($\varphi$)
4        $\varphi$ = Eq-Subs($\varphi$)
5        $\varphi$ = Eq-Rw($\varphi$)
6        $\varphi$ = Q-resolution($\varphi$)
7        if $\varphi \equiv$ TRUE  return $\varphi$
8        if $\varphi \equiv$ FALSE  return $\varphi$
9     while $\varphi$' $\neq \varphi$
10    return $\varphi$
```

(a) The algorithm of sQueezeBF.

```
0 function qubeEngine($\varphi$)
1     $\mu = \emptyset$
2     while (TRUE )
3        Propagate($\mu$,$\varphi$)
4        if (empty $\notin \varphi$)
5           $\mu.push$(Heuristic($\varphi$))
6        if (empty $\in \varphi$)
7           Backtrack($\mu$,$\varphi$)
8        else if ($\mu.top(B)$ == 0)
9           BuildPrimeImplicant($\mu$,$\varphi$)
10          Backtrack($\mu$,$\varphi$)
11       if (emptyClause $\in \varphi$)
12          return FALSE
13       if (emptyTerm $\in \varphi$)
14          return TRUE
```

(b) The algorithm of qubeEngine.

```
0 function Propagate($\mu$,$\varphi$)
1     start:
2        while (($l = \mu.next(B)$) $\neq$ 0)
3           ResolveBinaries($l$,$\mu$,$\varphi$)
4           if (empty $\in \varphi$) return
5        while (($l = \mu.next(S)$) $\neq$ 0)
6           Subsume($l$,$\varphi$)
7        while (($l = \mu.next(N)$) $\neq$ 0)
8           ResolveNaries($l$,$\mu$,$\varphi$)
9           if (empty $\in \varphi$) return
10          if ($\mu.top(B) \neq$ 0) goto start
11       while (($l = \mu.next(P)$) $\neq$ 0)
12          Search4pure($l$,$\mu$,$\varphi$)
13          if ($\mu.top(B) \neq$ 0) goto start
```

(c) The algorithm of propagate.

Figure 2. Main Algorithms in QuBE7. φ is the QBF being solved, whereas μ stands for the current assignment.

Q-resolution (line 6) can remove an existential variable x by replacing the sets of clauses S_x (where x occurs positively) and $S_{\overline{x}}$ (where x occurs negatively) with the set $S_x \otimes S_{\overline{x}}$ obtained by resolving on x all the pairs of clauses from S_x and $S_{\overline{x}}$, resulting in an equivalent problem. This is executed only when the total amount of literals in the resolvents ($|S_x \otimes S_{\overline{x}}|$) is less than the total amount of literals in the antecedents ($|S_x| \cup |S_{\overline{x}}|$). This technique was introduced for QBF in Quantor [1].

2.2 Search-Based Solver qubeEngine

QuBE7 solving engine is based on the QDLL algorithm [2], and inherits the techniques developed for the search-based solver included in QuBE6, such as pure/don't care literals detection, non-chronological backtracking and learning both for conflict and solution analysis [8]. Data structures for unit propagation and pure literal detection are "lazy" as described in [5], but differs from that of QuBE6 as it stores binary and n-ary constraints (from now on we will refer to clauses and terms as constraints) separately. Watched data structures are split as well. Further, data structures and algorithms are designed to improve data locality, in particular the loop executed in propagate (see Fig. 2(c)) performs the most expensive operations (e.g. unit propagation on n-ary constraints or detection of pure literals) when no cheap operation (e.g. unit propagation on binary constraints) is possible. To this purpose, the assignment stack μ keeps an index to the last literal assigned by each routine, that can get the literal they still have to propagate by using μ's method *next* passing as argument B, S, N or P respectively for unit propagation on binary constraints, subsumption of the n-ary clauses in the input QBF, unit propagation on n-ary constraints, and pure literals detection.

The main loop (Fig. 2(b)) takes as input the formula φ, and calls sequentially the following functions:

Propagate (line 3), which simplifies φ until no further steps are possible or an empty constraint is found (see below for a more detailed description).

Heuristic (line 5), which picks an unassigned variable according to a scoring function, and put that on the assignment stack. The score is computed as in QuBE6, where it depends on the number of constraints where the variable occurs.

Backtrack (line 7), which is called in case an empty constraint was found. The procedure to calculate the asserting constraint is an extension to QBF of that of MiniSat [4]. The idea is that, starting from an empty clause (resp. term), only the existential (resp. universal) literals can be resolved, and the Unique Implication Point must be existential (resp. universal) as well. In practice, we keep the highest decision level d of the existential (resp. universal) variables we met while backtracking, marking as unassigned the universal (resp. existential) variables and resolving the existential (resp. universal) variables till we have only one existential (resp. universal) literal l having level d in the current "asserting" clause (resp. term) X: In the problematic case in which the asserting clause (resp. term) X contains also an unassigned universal (existential) literal occurring to the left of l in the prefix (this is possible only if l was assigned as a unit), we cannot use X as asserting clause (resp. term) for assigning l as unit, and thus we keep backtracking resolving l.

BuildPrimeImplicant (line 9), which is called in case there is no empty constraint and there is not a new variable to propagate. It builds a prime implicant according to the procedure described in [8], afterwards the procedure Backtrack is called (line 10).

If any constraint in the formula is still empty, meaning that the conflict analysis computed either a 0-length clause or term, the function returns the satisfiability value of the given formula (lines 12 and 14).

In Propagate (see Fig. 2(c)) the literals in the assignment stack are used to detect new unit and pure literals in a lazy way. This means that unit propagation on binary constraints is always performed first when a new literal is assigned (line 3). If this propagation does not make any constraint empty, the assigned literals are used to subsume the clauses in the input QBF (line 6). Then constraint propagation is done on the n-ary constraints (line 8): if an empty constraint is found, the procedure terminates, or if a new unit literal is added to the stack it goes back to do BCP on binary constraints. Only at this point pure literals are detected and propagated (line 12), causing again unary propagation on binary constraints (assuming at least one literal is put on the stack by Search4pure at line 12).

2.3 Command-line Options

QuBE7 available options are:

`-ss`	: enables SelfSubsumption resolution
`-qr`	: enables Variable Elimination by Q-Resolution
`-es`	: enables equivalence substitution of AND/OR gates
`-3e`	: enables equivalence substitution of XOR gates
`-er`	: enables equivalences rewriting
`-all`	: enables all the above preprocessing techniques
`-noprepro`	: disables sQueezeBF, and gives the input formula to qubeEngine
`-solve`	: solves by using qubeEngine

Invoking QuBE7 without options produces a QBF resulting from the application of Simplify to the input QBF. For determining the value of the input QBF with the best performances, the default options to use are `-all -solve`.

3. Experimental Analysis

As environment, we used a farm of 9 identical PCs, each with an Intel Core 2 Duo 2.13 GHz, 4 GB RAM, running GNU Linux Debian 2.6.18.5; the time limit was set to 1200 s and the memory limit to 2 GB. We compare QuBE7.0 with the solvers that participated the QBF Evaluation 2010 on the same pool of (568) fixed-structure QBF instances selected for the main track of the same evaluation [7]. In Table 1 are shown the results. For each solver (first column) are reported the number of problems solved within the given time and memory limits, and the cumulative time needed (resp. second and third columns). In columns 4 and 5 (resp. 6 and 7) are listed in the same way the number of SAT (UNSAT) problems solved and the time needed. aqme-10 is a multi-engine solver, and is the only one able to solve more problems than QuBE7.0. For more detailed informations about the other solvers, please refer to QBFLIB [7].

References

[1] A. Biere. Resolve and expand. In *Proc. SAT 2004*, **3542** of *LNCS*, pages 59–70. Springer, 2004.

[2] M. Cadoli, A. Giovanardi, and M. Schaerf. An algorithm to evaluate Quantified Boolean Formulae. In *Proc. of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 262–267, 1998. AAAI Press.

[3] N. Eén and A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Proc. SAT 2005*, **3569** of *LNCS*, pages 61–75. Springer, 2005.

[4] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. SAT 2003*, **2919** of *LNCS*, pp. 502–518. Springer, 2004.

[5] I. Gent, E. Giunchiglia, M. Narizzano, A. Rowley, and A. Tacchella. Watched data structures for QBF solvers. In *Proc. SAT 2003*, **2919** of *LNCS*, pages 25–36. Springer, 2004.

Table 1. Results on the QBFEVAL10 Testset.

Solver	Total		Sat		Unsat	
	#	Time	#	Time	#	Time
aqme-10	434	32091.1	184	15825.6	250	16265.5
QuBE7	**403**	44204.3	180	26342.4	223	17861.9
depqbf	370	21515.3	164	13771.8	206	7743.5
qmaiga	361	43058.1	180	20696.6	181	22361.4
depqbf-pre	356	18995.9	172	12453.8	184	6542.1
AIGSolve	329	22786.6	171	12091.5	158	10695.1
struqs-10	240	32839.7	109	13805.5	131	19034.2
nenofex-qbfeval10	225	13786.9	109	8241.9	116	5545.1
quantor-3.1	205	6711.4	100	4130.6	105	2580.7

[6] E. Giunchiglia, P. Marin, and M. Narizzano. QuBE6.x, 2008. www.star-lab.it/~qube.

[7] E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. www.qbflib.org.

[8] E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause/term resolution and learning in the evaluation of quantified Boolean formulas. *Journal of Artificial Intelligence Research (JAIR)*, **26**:371–416, 2006.

[9] E. Giunchiglia, P. Marin, and M. Narizzano. An effective preprocessor for QBF pre-reasoning. In *2nd International Workshop on Quantification in Constraint Programming (QiCP 2008)*, 2008.

[10] E. Giunchiglia, P. Marin, and M. Narizzano. sQueezeBF: An Effective Preprocessor for QBFs Based on Equivalence Reasoning. In *Proc. SAT 2010*, **6175** of *LNCS*, pages 85–98. Springer, 2010.

[11] S. Malik and L. Zhang. Conflict Driven Learning in a Quantified Boolean Satisfiability Solver. *Computer-Aided Design, International Conference on*, pages 442–449, 2002.

[12] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[13] D. Plaisted and S. Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation*, **2**, pages 293–304, 1986.

[14] G. Tseitin. On the Complexity of Proofs in Propositional Logics. *Seminars in Mathematics*, **8**, 1970.

[15] L. Zhang. On Subsumption Removal and on-the-fly CNF Simplification. In *Proc. SAT 2005*, **3569** of *LNCS*, pages 482–489. Springer, 2005.